TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ANNA SOKOLINSKAYA

EXPERIMENTAL PRE-PROCESSOR FOR ACTION-BASED COMPUTING

Master of Science thesis

Examiner: prof. Hannu-Matti Järvinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 3rd December 2014

ABSTRACT

ANNA SOKOLINSKAYA: Experimental pre-processor for Action-based computing
Tampere University of technology
Master of Science Thesis, 49 pages
June 2016
Master's Degree Programme in Information Technology
Major: Software Systems
Examiner: Professor Hannu-Matti Järvinen

Keywords: concurrent computing, action paradigm, programming language theory

The conventional programming paradigms were developed for the sequential model of computation. Concurrent computing is implemented through processes. Communication between processes is implemented manually, which leads to a number of problems. In response to this, the action-oriented paradigm was developed, as well as the tentative action language. This language features actions as the main entities of execution containing all the functionality of the program. Actions are not executed sequentially; instead the next action to be executed is selected non-deterministically by the scheduler, which is a part of the operating system.

Implementing the action-oriented approach thoroughly requires the development of special hardware as well as the specially designed operating system. As this is not yet possible, a simulation environment has been developed in order to test and explore the action-oriented paradigm. This environment requires the solutions to be written in C programming language, following the specific format understandable for the environment. This structure does not correspond to the structure of an action program, and the key entities of the action-oriented paradigm are hidden behind cumbersome C functions.

This thesis presents a possible implementation of the simulation environment that provides opportunities to explore and test the action paradigm in a more convenient and coherent way. The suggested environment is based on the existing framework. A particular action language based on XML and C has been designed as a part of this thesis. As the implementation of the proposed environment is process-oriented on the low level, this language allows the user to write the functionality of the solutions in C language while maintaining the outline of an action program. XML syntax is used for the proposed action language in order to emphasize the hierarchical structure of an action program and utilize existing software tools to check, inspect and visualize action programs.

A pre-processor has been implemented to translate programs written in the proposed language into C programs in a form accepted by the existing environment. This procedure allows to hide the implementation details of the simulation environment from the user. The pre-processor does not perform full translation. Instead it restructures the elements of the program to meet the requirements of the existing framework.

The outcome of this thesis is the demonstration system that provides the means to investigate the action-oriented approach without designing special hardware tools. The system is user-friendly due to the usage of conventional XML syntax and C language.

PREFACE

I would like to thank my supervisor Professor Hannu-Matti Järvinen for the opportunity to write this thesis, for his patience and care.

I am truly grateful to my wonderful family for their understanding, their love and support under all circumstances.

I would like to thank Tanya and all of my friends for all the happy moments we shared during these years.

Wherever I go, I will always remember Tampere as one of the most amazing places in the world. I feel thankful to Finland for this unforgettable experience.

Tampere, 26.04.2016

Anna Sokolinskaya

CONTENTS

1. INTRODUCTION

In the field of parallel computing, simultaneous execution is usually modeled with the help of processes. Processes run concurrently, possibly sharing access to the same objects. Implementing this concept requires certain additional efforts from the programmer. Communication between processes is implemented manually. A programmer should maintain not only the underlying logic of a program, but also take care of critical sections, mutual exclusion, and synchronization. Specific problems like starvation and deadlocks should be solved. This requires a lot of extra work besides implementing functionality of the program. Traditional way of dealing with these matters is techniques like semaphores, monitors and message passing. A developer should also be aware of the number of processes. So, concurrency has to be programmed explicitly in addition to the internal logic of the program.

In contrast, action-oriented paradigm suggests a completely different approach to concurrency. It focuses on actions rather than processes, where each of the actions is an atomic change of the system state. Different actions can be executed simultaneously if they access distinct variables. The scheduler, which is a part of an operating system, selects a set of actions for execution. As a result, mutual exclusion and synchronization are guaranteed by the scheduler. The problems of deadlocks and starvation do not exist in the action paradigm. All the functionality is implemented in the actions which can change the objects assigned to them. Hence, communication between processes is hidden from the programmer. The hardware and the operating system take care of concurrency.

The implementation of the action approach requires hardware support and operation system rewriting. As this is not yet possible, an experimental simulation system has been developed. It is based on Posix threads and is process-oriented on low level. This system provides the means for testing and allows to apply the action-oriented model to particular tasks. The environment takes care of communication between processes.

However, the existing simulation system has some major drawbacks. The action-oriented view is not explicit there. The programs demonstrating the action approach must have particular format, which requires additional work to implement simple tasks. A solution written in action language has to be transformed into a complex and cumbersome C program, which does not have a structure of an action program.

The objective of this thesis is to implement of a new demonstration environment based on the existing framework. This environment will allow to demonstrate action paradigm in a more clear and convenient way. Since current implementation is process-oriented and action-based language is used only to illustrate the paradigm, there is no need for a full translator. Programs can be written in some action language on base of C.

An action language in this form has several aspects that allow to believe that introducing XML syntax for designing such a language is beneficial. First, action programs have a hierarchical structure, which can be easily converted to XML. Second, full translation is not needed, as reorganizing the values of the elements is sufficient. Utilization of XML allows to use existing software to write programs, inspect them, and check them for correct syntax. Visual tools can be helpful to illustrate the internal structure of an action program. Additionally, an action program in this form can be validated against a schema describing the grammar of the action language. Furthermore, XML syntax is familiar to many programmers and therefore it will be easier to introduce the action-based language into the programming community. Finally, it is possible to use existing parsers from XML to C or many other languages, which would make it easier to create action languages based not only on C, but on almost any other popular language in the future.

Hierarchical structure of an action program is demonstrated on Figure 1. The entities of the action language are represented as XML elements, and the values of these entities are stored as the contents of these elements. The pre-processor accepts an action program in a form illustrated by Figure 1 and generates a C program to be executed in the existing environment.

Figure 1: *Hierarchical structure of an action program*

The thesis is structured as follows. Necessary background is provided in Chapter 2. First, the problems of parallel computing are identified. Second, action paradigm is introduced along with its benefits for the programming of concurrency. A tentative action language is described and a detailed example of an action-based program is provided. Chapter 3 describes the existing simulation system and the demonstration system to be implemented. The syntax and semantics of the proposed C-based action language is defined. Chapter 4 describes the implementation of the pre-processor and provides several examples of action systems designed in the new environment.

2. BACKGROUND

This chapter introduces the action paradigm as a possible solution for the problems raised by implementation of concurrent computing. The key features of the action-oriented approach are identified as well as the general structure of an action system. The chapter is organized as follows. The existing mechanisms of implementing concurrency and the consequent problems are presented in Section 2.1. The action-oriented paradigm is described in Section 2.2. The main concepts of the tentative action language are given in Section 2.3. Finally, a simple example of an action system is provided in Section 2.4.

2.1 Utilizing multiple processor system

First computers supported the computation of only one program at a time. Nowadays, the existence of multiple processor systems allows parallel execution of several computational tasks. Traditionally, concurrent computing is based on the notion of processes, each being executed sequentially. That leads to a demand of special regulations and mechanisms supervising the communication and synchronization between processes [15].

A process is cooperating if it affects or can be affected by other processes in the system. There are several reasons for enabling interprocess communication. First, information sharing is often necessary. The same resource, for example, a file or a device, may be needed by several processes. Typically a common resource can be accessed by a limited number of processes at a time (usually one process). Second, cooperating processes provide the means for computational speedup by dividing a task into subtasks to be run concurrently on multiple processing elements. Interprocess communication mechanisms are needed for cooperating processes to interact. The communication between processes can be implemented either by shared memory, or by message passing [15].

The necessity for the several processes to gain access to the shared memory leads to a notion of a critical section. A critical section is a block of code that can be executed only by one process at a time. A process can acquire or release a lock over the critical section. A good locking algorithm implementing this mechanism should satisfy several key properties [4]:

- Mutual exclusion. This property guarantees that two processes cannot acquire access to their critical section at the same time. This property ensures the safety of the system, preventing corruption of the data.

- Freedom from deadlock. If some process is attempting to acquire the lock over a critical section, then some process will succeed in acquiring the lock. This property ensures liveliness of the system, as at least one of the processes should be making progress.
- Freedom from starvation. Every process attempting to acquire the lock will eventually succeed in acquiring it. This property appears to be the most problematic.

Synchronization between processes and maintaining the data consistency is a crucial problem of parallel computing. Key primitives for maintaining these tasks include semaphores, mutexes and monitors.

A semaphore is essentially a variable of a special data type used for controlling the access to a shared resource [16]. A semaphore supports two operations, down and up. The down operation waits for the value of a semaphore to be positive and decrements it. The up operation increments the value of a semaphore which results in completing the down operation by some process. Down and up are indivisible atomic operations. A simplified version of a semaphore is called mutex. Mutexes are intended only for managing mutual exclusion on a shared resource. A mutex is a variable that can be locked or unlocked.

A monitor is a high-level synchronization primitive, containing a collection of procedures and variables arranged in a module. Processes may access procedures in a monitor, but not the internal variables and data structures. Only one process can use a monitor at a given time. The compiler guarantees mutual exclusion on monitor entries.

The drawback of both semaphores and monitors it that these mechanisms were designed for processes that must have access to the common memory. If a distributed system with several CPUs is used, these primitives cannot be utilized.

Message passing represents another way of interprocess communication [16]. The mechanism of message passing allows the processes to communicate without sharing a region of memory. The basic primitives of message passing are *send(message)* and *receive(message)*. A communication link must be established between processes. There are several possibilities for the logical implementation of these primitives.

The communication may be implemented in a direct or indirect way. Direct communication implies that processes should explicitly specify a sender or a receiver in the communication. In this case a communication link is established between a specific pair of processes. Symmetric scheme of addressing requires both the sender and the receiver to be named. In the asymmetric variation, only the sender should state the receiver, and the receiving process is not expected to know the sender. The main drawback of direct communication is the restrained ability for modularity.

Indirect communication features ports, or mailboxes, as primitives for placing or retrieving messages. Two processes can communicate by using a shared mailbox. Every process can access a number of different mailboxes. Each communication link in this case is associated with a mailbox.

Message passing can be synchronous or asynchronous. In the synchronous variation of the send operation the sending process is blocked until the message is received. In the asynchronous variation, the sending process proceeds after sending a message. Similarly, synchronous version of the receive operation implies that the receiver is blocked until the message is received, and in the asynchronous version the receiver can get a null or a message. If both send and receive operations are synchronous, the variation of message passing is called rendezvous.

In order to implement the message passing mechanism, buffering is required. Messages are typically stored in a temporary queue of varying length. First possibility for implementing a buffer is a zero capacity queue, causing the sender to be blocked until the message is received. Another possible option is a queue of finite capacity, which allows several messages to be stored in the queue. Finally, the queue can have unbounded capacity, which empowers storing any number of messages.

The concept of message passing raises additional concerns. A message can be lost, a message should not be received several times, and authentication of processes should be supported.

Managing the synchronization and communication between processes requires sufficient efforts from the programmer. Though parallel computing tremendously accelerates the speed of the program, it consequently burdens the programmer with additional concerns about manipulation of processes. K. Chandy and J. Misra indicate in [1] the necessity of separation between core problem, hardware and language. They emphasize that the first concern should be to design a solution to the problem. The details of implementation on a given architecture should be considered separately. Another reason for reassessing the approach to the parallel computing is reusability. If a program is designed for a specific number of processors, subsequent modifications will be expensive. A possible solution would be to suggest an execution model, which allows to use a program on a variety of architectures without altering it.

Joint actions approach and Unity model were suggested in need of a theoretical model for programming on a variety of architectures and applications. Several issues crucial for such a model are highlighted in [1]:

- *Nondeterminism*. Specifying little about the execution of a program certifies efficient execution on a specific architecture.

- *Absence of control flow*. Traditionally, programming languages were based on a sequential flow of control. Parallel programming introduced the concept of processes as multiple flows of control. Processes remained sequential entities, while concurrency appeared to be a "special case" of sequential programming. Concealing the notion of control flow from the programmer's point of view shifts the focus to parallelism as the general case of computation.
- *Synchrony and asynchrony*. A unifying theory for parallel computing should include both synchronous and asynchronous models of communication.
- *States and assignments*. State-transition system is proposed in [1] as a base for a theoretical model of parallel computing since it is a convenient way of representation used in various disciplines. State-transition model serves as a base for an action system as well [9].
- *Proof system*. The logic of Unity [1] provides a proof system for verifying the correctness of the program. While action-oriented approach does not have a proof system of its own, it is closely related to the temporal logic of actions [8]. This theory provides means for proving properties of reactive systems. An action system can be understood in terms of the temporal logic of actions.
- *Separation of correctness and complexity*. As the same program is intended to be used on different architectures, the complexity of the program may vary depending on the way of execution. However, the correctness of the program should hold regardless of the architecture.

2.2 Action-oriented paradigm

Action-oriented paradigm is based on the notion of joint actions, introduced by R. Kurki-Suonio in [13]. The specification language DisCo for reactive systems was designed based on this research [7]. Joint action approach is similar to the Unity model, suggested independently by K. Chandy and J. Misra in [1].

In the traditional approach, the processes are the active entities of parallel execution. In the action-oriented approach, process are viewed only as resources for the actions. The action-oriented model of execution described in [7] focuses on the joint actions, which processes should accomplish together. There is no need for communication primitives between processes. All cooperation is modeled with the help of actions and their enabling conditions. Therefore, action paradigm eliminates the concept of processes. All the data in an action-oriented model is accumulated into objects, while objects cannot contain any functionality.

The state of the action system is determined by the states of all objects existing in the system. Each object represents an instance of a particular class. An object encapsulates its local state, preventing alteration by other objects existing in the system. An object is essentially a collection of data. Unlike object-oriented paradigm, action-oriented model

does not include the notion of encapsulated methods. Instead of that, each object has a number of roles in the actions. For each object, its roles allows it to participate in particular actions.

The difference between the notion of objects in object-oriented and the one in action-oriented paradigms is illustrated by Figure 2. The left part of the figure corresponds to the object-oriented approach, where external caller is used to invoke a method. Methods are printed in black to emphasize that their contents are private. In contrast to this approach, objects in action-oriented paradigm can not contain any methods. The functionality of the program is fully contained within action bodies. On the right side of Figure 2, methods are printed white to show that they are implemented inside actions A and B. Action A has three participants, while action B has two. Both actions A and B implement method 1 of object C. An action has to be selected by scheduler to be executed, but not by an external caller, which is emphasized by using lines instead of arrows [5].

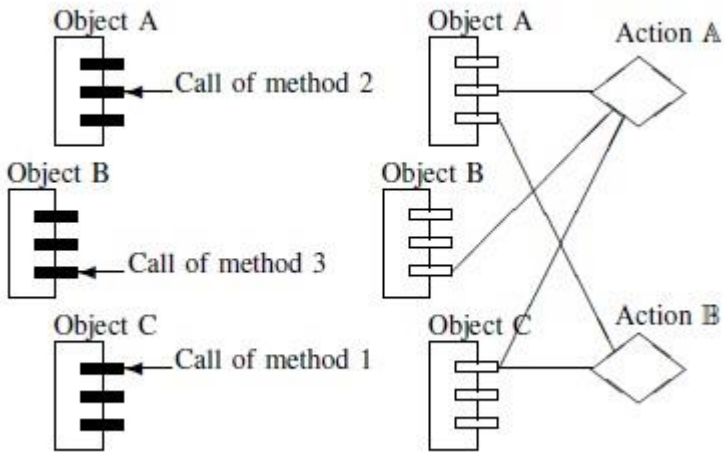

Figure 2: *Objects in object-oriented and action-oriented paradigms [5]*

Actions are atomic units of execution in the action-oriented paradigm. An action represents a state transition between two states of the system. An execution in action system consists of a number of steps, each of them corresponding to an action and representing a change between system states [11]. Figure 3 displays the execution of actions *<A1, A2, A3, ...>* corresponding to the state sequence *<s0, s1, s2, ...>*.

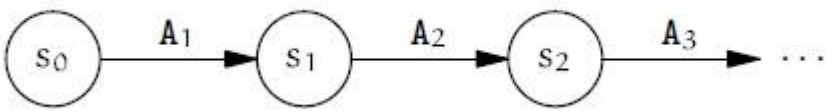

Figure 3: *Execution of actions [11]*

An action is either enabled or disabled in a particular state. Only enabled actions can be selected for execution. The first state of execution is the initial state of the system. An

execution of an action system is terminating if there exist a final state and infinite otherwise. Figure 3 illustrates an infinite execution, where *s0* is the initial state [11].

Each next step of the execution in an action system is chosen nondeterministically. Figure 4 illustrates an example of nondeterministic choice. Two actions are enabled in a state *s*, and both of them can be selected as a next step of execution [11].

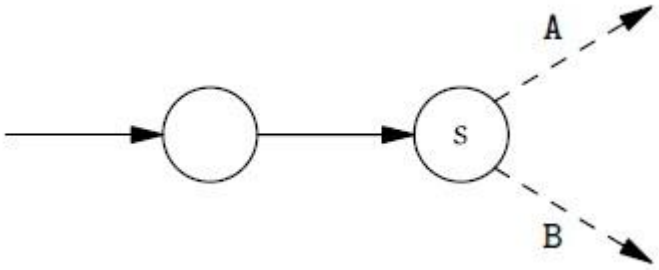

Figure 4: *Example of nondeterministic choice [11]*

The existence of nondeterministic choice between actions provides the means for implementing concurrent and distributed systems. If two actions have distinct sets of variables as their participants, they can be executed simultaneously. Formally, for two actions A and B, where V_A^a is the set of variables assigned to by the action A and V_A^r is the set of variables referred to only by action A, then simultaneous execution can be performed to the set S of distinct actions where (1) holds true. Objects can be considered instead of variables in (1) as well [11].

$$\forall A, B \in S, A \neq B: V_A^a \cap V_B^a = \emptyset \ \wedge \ V_A^a \cap V_B^r = \emptyset \ \wedge \ V_B^a \cap V_A^r = \emptyset \qquad (1)$$

Atomicity is an essential characteristic for actions. The execution of an action should be completed without interruption from other actions. Figure 5 illustrates a situation where actions *A1* and *A2* seemingly produce the same combined effect as action *A*. However, it leads to the appearance of an intermediate state *s'*. At this state some other enabled actions might exist, leading to new possible executions of the system. Atomicity of actions guarantees that the action model can be safely used in the distributed and concurrent systems [11].

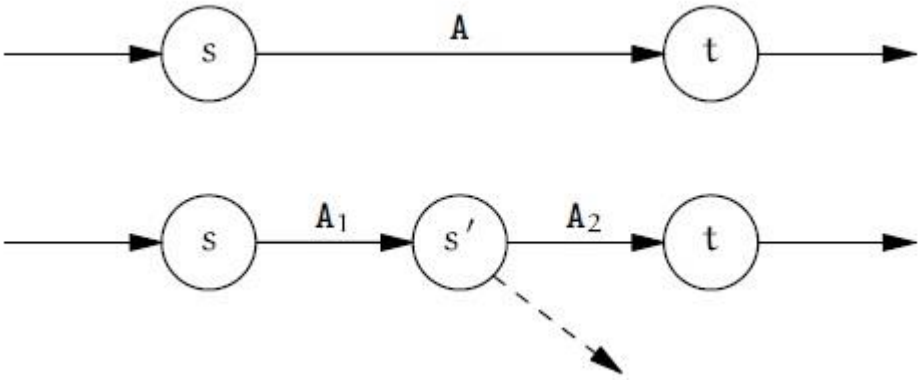

Figure 5: *Example of importance of atomic execution [11]*

Unlike traditional models, action-oriented paradigm does not extend the traditional sequential model of computation. The program counter does not exist. Any action enabled in the current state can be selected for execution. An action states what is done, while the process focuses on who does it [11].

The general structure of an action system is illustrated by Figure 6. An action system consists of *n* processing units and a scheduler which is a part of the operating system. To achieve the maximum efficiency, processing units should be connected with special hardware. All the actions existing in the system are contained in the action store. The scheduler selects a set of enabled actions from the action store. These actions should have distinct participants with each other as well as with other actions in the queue or in execution. Next, one of the processing units takes one action from the queue and executes it. If processing units do not have a common memory, action code and accessed objects are loaded into the local memory of this processor. In this case, altered objects are copied to common memory after the execution of the action. The executed action is returned to the action store [5].

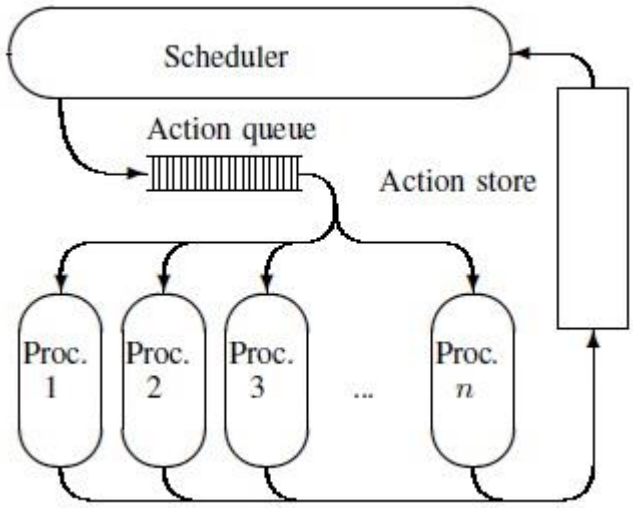

Figure 6: *General structure of an action system [5]*

The program written in action language will not be affected by changing the number of executing processes. Concurrent execution is supported by hardware and the scheduler, but not by the programmer himself. As the proper hardware for implementing action system does not exist yet, this model of execution has been explored and tested in a simulation environment built on top of a general purpose operating system, which is described in Section 3.1.

Action-oriented paradigm has been primarily enforced for the specification and design of reactive systems. However, if used as a low-level mechanism of computation, action-oriented model can allow the programmer to be focused on the particular computational task without explicitly programming concurrency features [5]. Action orientation provides a synchronization mechanism since synchronization conditions are expressed by action guards. The scheduler guarantees mutual exclusion, and the notion of critical sections does not exist anymore. Concurrency is handled by hardware and the scheduler instead of the programmer. Any number of processors can be used for the computational tasks without altering the program. The scheduler is responsible for managing their cooperation.

Although action paradigm has been analyzed to have significant benefits for the tasks of parallel computing, it has not been yet implemented for practical use. One of the reasons for that might be the fact that programmers are used to think in terms of sequential order of operations. Action paradigm requires effort to understand and adapt to it. Second, in order to experience the benefits of action paradigm, it has to be implemented on hardware level. A scheduler must be a part of the operating system, and implementing a scheduler requires cautious work [5].

2.3 Features of the tentative action language

The tentative language described in this section is based on the DisCo language and the language described in [5]. This description is not meant to be complete, as this language is used only as a basis for the language of the proposed simulation environment. The purpose of this description is to signify the main features of the action language to be simulated.

A program in the action language consists of three phases: description of classes, description of actions and initialization phase responsible for creating instances of these descriptions. Actions, classes and objects must have unique names in the action system. All parameters and participants within an action should be named differently.

All the data in the system is encapsulated into objects. Each object is an instance of a particular class. Any data structure can be a part of an object. A class is defined as follows:

```
class class_name is
    class_body
end;
```

Class body consists of a list of class variables and their initial values. Each variable definition takes the following form:

```
variable_definition = variable_name_list : type [ := expression ];
```

The optional *expression* part provides the default value of variable(s), which is evaluated when an object of the given class is created.

Example (adapted from [12])

The following example is used to illustrate the syntax of the action language. A more detailed example will be provided in Section 2.4.

The *Radio* class describes a simple radio, which is capable of transmitting and receiving messages. A radio can prepare and transmit one message at a time. A radio has two mutually exclusive modes: talking mode is used for transmitting messages and listening mode corresponds to receiving messages. An additional property state describes whether a radio is currently involved into communication or not.

```
class Radio is
    mode: (LISTENING, TALKING);
    state: (READY, ENGAGED);
    message: message_type;
end;
```

Communication is implemented through a channel, which has capacity for one message. A channel is either free or in the state of transmitting a message. The *Channel* class implements this entity.

```
class Channel is
    state: (FREE, BUSY):= FREE;
    message: message_type;
end;
```

Actions are the units of execution where all system functionality is contained. An action has an optional list of parameters, a set of participants that are engaged in the action in specific roles, a guard and a body.

```
action_definition = action name [(parameters_list)] is
    participant_description; { participant_description; }
when common_guard;
body
    statement; {statement; }
end;
participant_description = participant_class as participant_name: local_guard;
common_guard = boolean_expression;
local_guard = boolean_expression;
```

The list of action participants is non-empty. The participants are distinct, meaning that any object can participate in a particular action in one role only. Each participant belongs to one of the predefined classes. Each formal participant corresponds to some role in the action. When an action instance is created, an object is assigned to be the actual participant of the action.

An action can have an optional list of parameters of predefined types. When an instance of the action is created, actual values are assigned to parameters.

The action guard is a Boolean statement that is the prerequisite for the action to be enabled. Only enabled actions can be selected for execution. The action guard can refer only to the contents of the participants. The guard syntactically consists of the common guard and a set of local guards corresponding to the participants of the action. Each local guard refers to the contents of a single participant. The common guard refers to the contents of several participants. The separation of the action guard into these parts permits the scheduler to be implemented in a more efficient way. There is no need to evaluate the contents of the common guard if one of the local guards is false. The local guards are evaluated only if the contents of the participant has changed since the previous evaluation. Joint actions and DisCo language additionally included global part of the action guard, referring to any variable or object in the system. The implementation of this concept was difficult and inefficient, so this part of the action guard does not exist in this variation of the action language [5].

The body of an action contains the code to be executed when the action is selected by the scheduler. The body can refer to the participants of the action and not to any other object existing in the system.

In order to simulate the process of exchanging messages between radios, several actions are determined in the example. Actions *send* and *receive* are responsible for transmitting a message. A message that was transmitted through a channel can be received by any available radio in the listening mode. Action *prepare* simulates the process of producing a message. For simplicity, each radio tries to broadcast a message every time a certain timeout has passed. The same channel can be used by any number of radios. Action *digest* implements processing a message by the receiving radio.

Action *send* has no parameters and two participants of classes *radio* and *channel*. The common guard is empty and the local guards of the participants describe the prerequisites for the action to be enabled. The body of the action provides a set of statements to be executed if the action is selected by the scheduler.

```
action send is
    Radio as sender: mode = TALKING and state = ENGAGED;
    Channel as channel: state = FREE;
body
    channel.state := BUSY;
    channel.message := sender.message;
    sender.state := READY;
    sender.mode := LISTENING;
end Send;
```

Symmetrically to the send action, the receive action has no parameters and two participants: a channel and a receiving radio. The guard of the action requires that the channel is busy to be engaged in this action, and the receiver is listening and is not engaged in other communications.

```
action receive is
    Radio as receiver: MODE = LISTENING and state = READY;
    Channel as channel: state = BUSY;
body
    channel.state := FREE;
    receiver.state := ENGAGED;
    receiver.message := channel.message;
end;
```

Action prepare corresponds to the process of preparing a message every time a timeout has passed. It has two parameters: a message to be sent and the value of the timeout.

```
action prepare (m: message_type, gap: seconds) is
    Radio as sender: mode = LISTENING and state = READY and timeout(gap)
body
    sender.mode := TALKING;
    sender.state := ENGAGED;
    sender.message := m;
end;
```

Action digest simulates the process of processing a message by the receiver. Similarly to the prepare action, action digest has only one participant of class radio.

```
action digest is
    Radio as receiver: mode = LISTENING and state = ENGAGED;
body
    state := READY;
end;
```

The objects and actions are created in the initialization phase. Each object is a named instance of some class, while each action is an unnamed instance of some action descriptor. The static participants and parameters are assigned to actions during initialization phase. The initialization code must be sequentially executed in order to invoke the action system. Additionally, new actions and objects can be created and deleted in the run-time mode.

The initialization code for the example action system is provided below.

```
walkie1, walkie2 : Radio;
channel1: Channel;
create prepare (“ping”, 10, walkie1);
create digest (walkie2);
create send (walkie1, channel1);
create receive (walkie2, channel1);
```

The first two statements create two objects of *Radio* classes and one object of class *Channel*. The properties of these objects are assigned default values. The next statements create several actions of different types. Objects created above are assigned to be participants of these actions. Constant values of integer and string types are assigned to be parameters of the action from the *Prepare* descriptor.

2.4 Example of an action system

The following example is adapted from [5] to illustrate the action paradigm. This example will be further implemented in the existing and proposed demonstration environments.

This example models a crossing of two roads, one going from north to south and the other from west to east. Each direction has two separate lanes, one of which is used to go forward or turn right, and the other is used to turn left. Each lane has a corresponding traffic light. Each traffic light has three possible lights: red, yellow and green.

Each lane has a unique name, where the first letter stands for the approaching direction, while the second and possibly the third letters stand for destination. Two lanes labeled on Figure 7 go from south to west (SW) and from south to north and east (SNE).

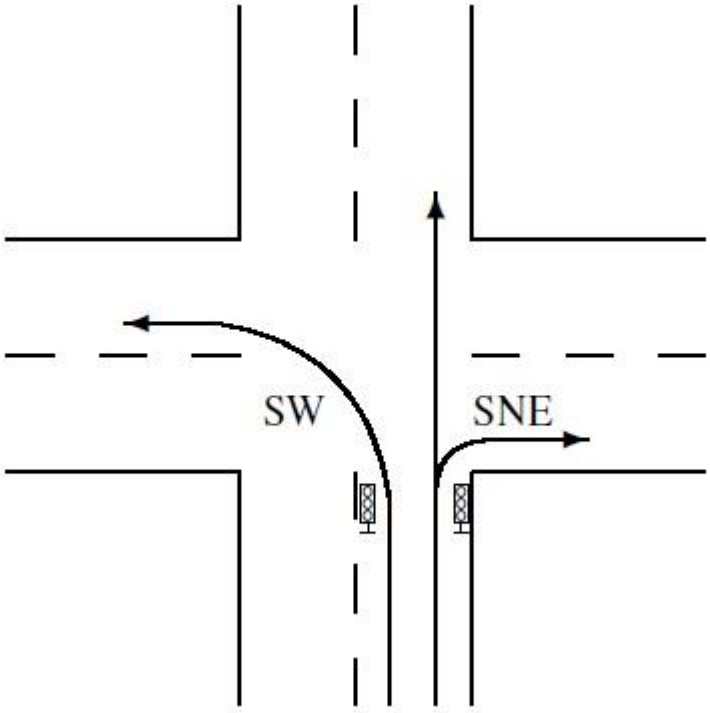

Figure 7: *Illustration of the crossing problem [5]*

The main abstractions in this example are a lane and a traffic light. As each lane has its own traffic light, it can be set to green or yellow only when the corresponding lane is safe to drive. A lane is considered safe when all the lanes crossing it have their corresponding traffic lights set to red.

According to the traffic regulations, certain pairs of lanes can be utilized simultaneously. Traffic lights corresponding to these lanes can be set to green at the same time. One possible way of forming these pairs is to combine NSW and SNE, WES and EWN, NE and SW, WN and ES. Each direction appears once in this set, therefore launching traffic on each of these pairs will result in exploiting every possible direction.

The classes corresponding to the key entities of this example can be designed as follows:

```
class Lane is
    safe: Boolean := false;
end;
class Traffic_light is
    lamp: (RED, YELLOW, GREEN) := RED;
end;
```

The *Lane* class has only one property, which is a Boolean variable indicating whether a lane is safe for driving or not. Class *Traffic_light* represents a traffic light and has one property which states its current color.

An auxiliary class is used for preventing car collisions. Class *Control* is responsible for alternating between system states and ensuring that only safe pairs of lanes are used simultaneously. This class features one property *state* that can be assigned one of four values, each corresponding to a particular pair of lanes. Alternating these states ensures that all lanes are eventually appointed safe.

```
type Safe_pairs is
    (NSW_SNE, NE_SW, WES_EWN, WN_ES);
class Control is
    state: Safe_pairs := NSW_SNE;
end;
```

Actions identified in the model include actions responsible for changing colors of traffic lights as well as actions implementing the regulation of traffic on the lanes.

Action *set_green* sets a particular traffic light to green. This action has two participants, one of class *Lane* and one of class *Traffic_light*. The participating lane has to be safe and the corresponding traffic light has to be set to red. Under these conditions the action is enabled.

```
action set_green is
    Lane as lane: safe;
    Traf_light as light: lamp=RED;
body
    light.lamp:=GREEN;
end;
```

Action *set_yellow* has one participant of *Traffic_light* class. This action is enabled if the light of the participating traffic light is green. Additionally, this action has a parameter green_on indicating the duration of green light. The value of the parameter is assigned when the action is created. Timeout is a Boolean function, which return true if the time stated by its argument has passed since the previous update on the object.

```
action set_yellow(green_on: seconds) is
    Traf_light as light: lamp=GREEN and timeout(green_on);
body
    light.lamp:=YELLOW;
end;
```

Action *set_red* is responsible for assigning red color to a traffic light. Similarly to *set_yellow* action, it has one participant representing a traffic light and a parameter stating the duration of the previous color (yellow).

```
action set_red (yellow_on: seconds) is
    Traf_light as light: lamp=YELLOW and timeout(yellow_on);
body
    light.lamp:=RED;
end;
```

Action *free_lane* verifies that a lane is safe to use. It has two participants: a lane and a traffic light, and one parameter stating how many seconds should pass after red light is

lit until the lane is guaranteed to be free. The prerequisites for enabling this action are the safety of the lane and the red color of the traffic light. The action waits *margin* seconds until there are no cars on the lane, then marks it as unsafe.

```
action free_lane (margin: seconds) is
    Lane as lane: safe;
    readonly Traf_light as light: lamp=RED and timeout(margin);
body
    lane.safe:=false;
end;
```

Action *reserve_lane* is used to ensure that all the lanes are equally utilized in the system. It has two parameters that correspond to the old and new system states. The participants of this action are the control object, a pair of lanes that have just been assigned green light and a pair of lanes to be marked safe for driving.

```
action reserve_lane (current, next: Safe_pairs) is
    Control as cont: state=current;
    readonly Lane as old_1, old_2: not safe;
    Lane as new_1, new_2: not safe;
body
    cont.state := next;
    new_1.safe, new_2.safe:=true, true;
end;
```

All the actions and objects in this example are created during initialization phase before the action system starts. As participants of each action are assigned static values during initialization, the common guards are not needed for describing the relationship between participants. When the system starts, it is intended to execute forever.

```
Initially
    nsw, sne, ne, sw, wes, ewn, wn, es: Lane; // Creates lanes
    tl_nsw, tl_sne, tl_ne, tl_sw, tl_wes, tl_ewn, tl_wn, tl_es: Traf_light;
    control: Control;
    // Some constant values
    green_on, yellow_on: seconds:= 20, 5;
    margin: seconds:=8;
    // Create actions for a lane
    create set_red(yellow_on, nse);
    create set_yellow(green_on, nse);
    create set_green(nse, tl_nse);
    create free_lane(margin, nse, tl_nse);
    // etc. for each lane
    create reserve_lane(NSW_SNE, NE_SW, nsw, sne, ne, sw);
    // etc. for each safe pair
end initially;
```

3. TECHNICAL DESCRIPTION

This chapter provides the technical description of the proposed simulation system. This system is designed for exploring and testing the action-oriented approach by creating and executing programs written in a particular action language based on XML and C. This chapter has the following structure. Section 3.1 describes the existing simulation system and its drawbacks that led to the proposition of the new simulation environment. The key features of the action language for the system are identified in Section 3.2. The detailed description of the proposed language as well as the general structure of the suggested demonstration environment are given in Section 3.3.

3.1 The existing simulation system

Action-oriented paradigm suggests a new approach to designing programs. Concurrency primitives are completely hidden from the programmer's point of view. However, the implementation of a low-level action-oriented system requires hardware support and the development of a special operating system, which is a laborious task.

As the action-oriented approach contradicts to the casual way of thinking, adapting to it is a challenge in itself. A simulation experimental environment was developed to test an action-oriented paradigm. The simulation environment works on top of a general-purpose operating system and provides the interface to apply the action-oriented approach to a particular task. The actual implementation is sequential and process-oriented, but these technicalities are hidden from the programmer.

The terminology used in the simulation system differs from the conventional terminology adopted from sources such as [5][9] and described in Section 2.3. The reason for it is the necessity of distinction between action definitions and their instances. Both of these concepts are referred to as “actions” in the conventional terminology. In order to prevent confusion, action definitions are referred to as “action descriptors” in this chapter and in the description of the proposed simulation system in Section 3.3. The instances of these descriptors are termed “actions”. For purposes of consistency class definitions are called “class descriptors” in these chapters, while their instances are labeled “objects”.

3.1.1 Additional features of the action language implemented in the environment

Besides the functionality of an action system described in Section 2.2, the simulation environment implements several additional features. The timing properties are introduced to delay the execution of actions. An action descriptor can provide a value specifying a particular rate, in which action instances are executed. In this case a descriptor can determine the relative deadline - the interval within which the execution must take place. If the rate is not provided for a descriptor, indicating a relative deadline is meaningless. Assignment of the rate and relative deadline establishes new rules for enabling an action; it implies that the common guard is always true.

Furthermore, the execution of an action can be delayed with regard to individual participants. To ensure that the action will adequately respond to the change in the participating object, local timeouts and deadlines are introduced. A local timeout specifies the first possible time of execution after the change in the participating object, while a local deadline provides the latest time when the action is invoked. The timing starts when the local guard of the corresponding object is verified to be true. These values are provided independently; if the local timeout is not specified, the local deadline still indicates the end of the time interval.

Sometimes an action should be executed only once after the participating object has been changed, even if the guard still remains true. This can be implemented by introducing a new property for an action descriptor, which indicates that each instance of this descriptor is enabled only if the participating object has just been changed. When this property is true for several participants of the same action descriptor, its instances are enabled in case any of these objects has been updated since the previous execution.

3.1.2 Interface description

The existing environment does not provide opportunities for compiling programs written in the action language. A program is written in C, and must have a special structure in order to invoke an action system. The environment is implemented with the help of Posix threads, and it can be used under Windows and Linux operating systems.

The interface supporting the simulation of an action system is presented in a C header file [6]. This interface provides functions that correspond to the logical structure of an action program.

All the data is stored in objects, and all the functionality is implemented through actions. The number of participants of an action is limited to 31. The number of executing threads is provided explicitly, minimum being 1 and maximum 10.

In order to start the action system, a *runner* function with a particular signature is called. Its first parameter specifies the number of executing threads, while the second determines the *application* function:

```
int runner (uint threads, application application_init);
```

The interface of the environment provides functions for creating class descriptors, action descriptors, objects and actions. These entities are created within the application function.

```
typedef void (*application) ();
```

To create a class, its name and size are provided. Optional arguments include functions used to initialize, copy and destroy objects of the given class:

```
Class_descriptor create_class(char *name,
    uint size,
    Initial_object initially,
    Copy copy,
    Destructor destroy);
```

Action descriptors are created using the following function:

```
Action_descriptor create_action_descriptor (char *name,
    Actionfunction body, Guardfunction common_guard,
    Timings timing, Mask read_only,
    uint parameter_count, uint participant_count, ...);
```

Functions representing the action body, the common guard and local guards for each of the participants are created beforehand. Any parameter representing a guard can be equal to NULL, in which case the guard will be assumed to be true.

Parameter timing of the *create_action_descriptor(…)* function provides the address to the timing function:

```
typedef Timings_type (*Timings) (uint parameters[]);
```

This function returns a structure of the following type:

```
typedef struct Timings_type {
    uint rate;
    uint relative_deadline;
    uint local_timeouts[MAX_PARTICIPANTS];
    uint local_deadlines[MAX_PARTICIPANTS];
    bool one_shot[MAX_PARTICIPANTS];
} Timings_type;
```

The fields of this structure specify the values of rate and relative deadline for the action descriptor, as well as the set of local timeouts and local deadlines for each participants. The array *one_shot* provides a Boolean value for each of the participant. Setting it to be

true means that the action is executed only once after the value of the corresponding participant has been changed. The meaning of these terms is described in Subsection 3.1.1. Only the local timeouts for the participating objects are implemented in the current version of the simulation environment. Other timing parameters can be assigned, but they will have no effect on the execution.

Parameter *read_only* of the *create_action_descriptor(…)* function provides a mask which indicates what participants are read-only. This parameter can be used to increase efficiency, but this is not implemented in the current version.

Parameter *parameter_count* specifies a number of action parameters. Parameters are converted to integer values in the current implementation of the simulation environment.

Parameter *participant_count* determines the number of participating objects, and the subsequent parameters provide addresses of local guard functions for each participant. Any of the local guards can be set to NULL, meaning that it is always evaluated true.

The function implementing the action body has the following type:

```
typedef void (*Actionfunction)(uint parameters[],  // Action parameters
    Objectpointer p[],  // Action participants
    Mask *unchanged,  // If any of the participants has been updated
    Mask *deleted,    // If any of the participants has been deleted
    Actionpointer runner);    // myself
```

All the objects of an action are pointers to the instances of Object type. Their order is the same as in the corresponding *create_action_descriptor(…)* call. The order of the action parameters is established likewise. Parameter *unchanged* of the function call provides the mask stating which participating objects have been modified by the action. Similarly, parameter *deleted* determines, which participants have been deleted. These values are assigned inside the body function.

Parameter *runner* points to the action being executed by this body. This value is needed in case actions or objects are created of deleted inside this action body.

Functions creating class and action descriptors have return values of types *class_descriptor* and *action_descriptor* respectively. These values are used in the initialization phase. The initialization phase is implemented by invoking functions that create objects and actions from the descriptors defined previously.

All the objects in the system belong to the type *Object*. They are created with the function *create_object(…)*, which returns the pointer to the object.

```
typedef struct Object *volatile Objectpointer;
Objectpointer create_object (Class_descriptor);
```

Actions are created with *create_action* function:

```
void create_action( Actionpointer runner, Action_description description,
    ...
    //uint parameters // This is repeated parameter_count times, 0 or more
    //Objectpointer // this is repeated participant_count times, 1 or more
);
```

Parameter *runner* specifies the calling action. If the action is created from the application initialization function, this parameter is set to be NULL. If the action is created from another action, this parameter indicates the executing action. Action descriptor is provided next, determining the descriptor, instance of which is being created. Next parameters give the actual values of action parameters. The number of parameters is equal to *parameter_count* specified by the corresponding call of *create_action_descriptor(…)*. Subsequently, pointers to the participating objects are provided as parameters. Their number is equal to the corresponding value of *participant_count*.

Actions can be created not only during the initialization phase, but also inside other actions. In this case function *create_action(…)* is called from the action body with the pointer to the executing function as the *runner* parameter. In the same way, action can be deleted inside the action body. Usually the executing function deletes itself, but other actions can also be deleted. Deletion is performed by calling the following action:

```
void delete_action(Actionpointer runner);
```

Another function provided by the environment permits to duplicate objects inside the action body:

```
void duplicate_objects(Actionpointer runner, Objectpointer new_objects[],
    uint count, ...);
```

The array used for the new objects is created beforehand.

3.1.3 Execution of the environment

The interface of the environment is provided in the *actionrun.h* header file. The implementation of this interface is contained in the *actionrun.c* file. To compile this file, the following command may be used:

```
gcc -std=c99 -o actionrun.o actionrun.c
```

An action program must have the following directive in order to use the environment:

```
#include “actionrun.h”
```

After the program is executed, analytical information is printed. It can look as follows:

```
Action count 0, create_count 0 mutex called 48
Creation part 44, actions found 48 action null 17283 queue actions 48
```

```
Action lock locals valid value mode
Executor 0: 35 rounds (27 run, 8 deletes)
Executor 1: 13 rounds (5 run, 8 deletes)
```

The *action count* value indicates the number of actions still existing after termination. The value of *create_count* is equal to the number of unfinished creations of actions. It should normally be 0. The *mutex called* value tells the number of times when critical section was entered by executors. The value of *creation part* shows how many times actions were finalized. It is 0, if actions do not create new actions or objects. The value of *actions found* indicates how many times an action was chosen for execution by the scheduler. The value of *action null* tells how many times there were found no applicable functions. The value of *queue actions* is equal to the number of actions put in the queue by the scheduler and should be the same as the value of *actions found*.

Next line (*Action lock…*) is a header for the table of errors. If no table is printed afterwards, the execution was successful. Otherwise, one or more actions have not been unlocked during the execution. Their identifiers are displayed in the table as well as the information about their state.

Next several lines of the output provide the information about particular executing threads. The number of *rounds* for each executer shows how many times it was given an action to execute. The sum of *rounds* should be equal to the *queue actions* value. After each value of *rounds* more detailed information in the parentheses tells how many times the corresponding thread actually executed an action, and how many times it performed a deletion.

3.1.4 Reasons for creating a new system

While the existing environment successfully simulates the action-oriented approach upon the general purpose operating system, its significant drawback is the lack of clarity. Although the environment provides the tools to test and explore the action paradigm, example programs do not follow the structure of a typical action program. It takes time and effort to understand how the cumbersome example programs written in C implement the action approach. As separate functions are provided for each guard, body or timing settings, the overall structure of the program becomes unclear. The following code fragment illustrates the implementation of the essential functions for a single action *set_green* from the example described in Section 2.4:

```
bool guard_green_local_lane(Objectpointer object) {
    struct Lane *lane = (struct Lane*) object_data(object);
    return(lane->safe);
}
bool guard_green_local_light(Objectpointer object) {
    struct Light *light = (struct Light*) object_data(object);
    return(light->lamp==RED);
}
```

```
void body_green(uint *parameters, Objectpointer *objects, Mask *unchanged,
Mask *deleted, Actionpointer runner) {
    struct Lane* lane = (struct Lane*) object_data(objects[0]);
    struct Light* light = (struct Light*) object_data(objects[1]);
    light->lamp=GREEN;
    printf("Lane %s set GREEN at %u\n", names[light->id], (uint)time(NULL));
    *unchanged = 1; *deleted = 0;
}
void application_body() {
  ...
  Action_descriptor green =
    create_action_descriptor("Green light", body_green, NULL, NULL,
        0, 0, 2, guard_green_local_lane, guard_green_local_light);
    ...
    for (uint i = 0; i < 8; i++) {
    create_action(NULL, green, objects[i+8], objects[i]);
    }
}
```

The preceding code demonstrates that in order to create an action program in the existing simulation environment, a programmer should be well versed in the rules and regulation of the provided interface. Besides, a considerable amount of practice is needed for a user to understand the meaning of the existing program. These considerations have led to the idea of implementing a new demonstration environment which will simulate action execution in a more transparent way.

3.2 Action-based language on base of C language

The existing simulation system provides mechanisms to test and explore action paradigm, but does not provide the suitable interface. The idea behind this thesis is to create a new, user-friendly simulation environment on top of the existing one. As the simulation environment is process-oriented at low level, there is no need to implement the action language in details, since its main features include the structure of actions and objects. The actual implementation of the action bodies and object data structures is not important to test the concept. Under these circumstances, implementing a complete translator would be redundant.

The proposed solution is to implement a new simulation environment based on the old one in order to further analyze and test the action-oriented paradigm. Programs in this environment can be written in a particular “action language” on base of C language. The pre-processor is needed to translate programs from this action language into C program understandable for the existing environment. Figure 8 illustrates the relation between the existing simulation environment and the proposed C-based language pre-processor.

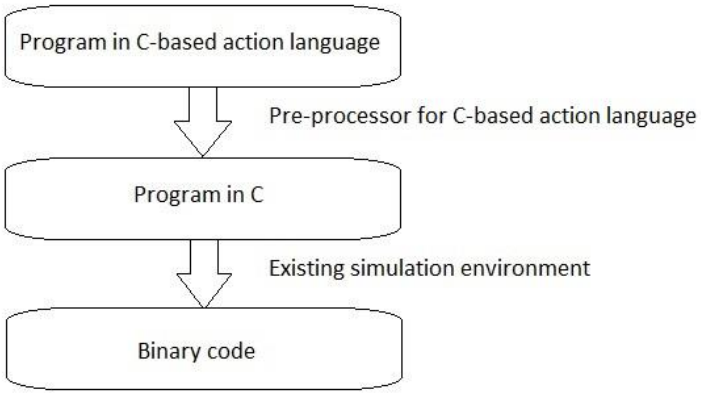

Figure 8: *Structure of the proposed simulation system*

A program in the C-based action language is intended to follow the structure of the action program described in Section 2.3. It consists of class descriptors, action descriptors and initialization phase. The logical expressions, action bodies and internal class data structures are to be written in C. This approach allows users to organize their programs according to the action paradigm while eliminating the concern of learning a new language.

A pre-processor is needed to translate an action program into the C program understandable by the existing environment.

The initial idea of the syntax for the C-based action language was to introduce keywords and specific brackets to separate different parts of an action program. A possible syntax of the C-based action language is demonstrated by Program 1.

```
CLASS <<< LANE >>> AS <<< struct Lane >>>
CLASS <<< LIGHT >>> AS <<< struct Light >>>
CLASS <<< CONTROL >>> AS <<< struct Control >>>
ACTION <<<set_green>>>
ON <<<LANE>>> AS <<<lane>>> IS <<<safe==true>>>
ON <<<LIGHT>>> AS <<<light>>> IS <<<lamp=RED>>>
WHEN <<<true>>>
BODY <<<
light.lamp:=GREEN;
>>>
```

Program 1*: Possible syntax for an action program*

However, this approach has several drawbacks. First, such a syntax looks ponderous and difficult to read. Second, a programmer might be reluctant to adjust to a new syntax.

An action program in the form of Program 1 was found to have certain distinctive features. Each program appears to be a set of pairs “attribute” - “value”, where attributes are code words and values are valid statements of C-code. The hierarchical structure of

an action program is emphasized in Program 1 by applying bold font for “attributes”, while “values” are enclosed in triple angle brackets.

The observed hierarchical structure of an action program led to the proposition of XML syntax for the C-based action language. The benefits of this approach are explored in the following section.

3.3 Proposed demonstration system simulating the use of the action language

3.3.1 Benefits of XML syntax for an action language

The Extensible Markup Language (XML) was developed in guidance of World Wide Web Consortium (W3C) in 1996. The design goals underneath this language included unified syntax suitable for straightforward use over the Internet, simplicity in usage, concise design, clearness for a human reader [3]. XML defines a straightforward syntax standard for maintaining compatibility between different systems. W3C XML Schema allows to specify the desired structure of an XML document [18].

XML essentially provides the means to store structured information, containing both data and the indication of its meaning in terms of certain structure (grammar). The XML specification determines a unified way of including markup into documents [17]. There are no predefined tags; all the necessary rules and regulations can be described through an XML Schema document.

The popularity of XML led to the development of multiple software tools for operating XML documents [14]. This software provides user-friendly interface for creating and editing XML documents as well as validating them against their schemas. Moreover, software tools can allow the programmer to traverse XML nodes in a form of a tree and export their content to other formats. Available software includes both open-source and proprietary solutions, thus providing affordability.

XML proposes a clear, straightforward and well-known syntax standard for accumulating hierarchical structured data. There are several underlying reasons to introduce this syntax for the C-based action language instead of the superficial syntax illustrated by Program 1:

- Action programs have a clear hierarchical structure, which can be directly expressed in XML.
- Full translation from an action language to C is not required, since all the meaningful parts of action programs are already written in C code. The remaining entities include identifiers and parameter values that are easy to convert to C.

Therefore, reorganizing and processing the element values become sufficient to acquire a C program that is comprehensible to the existing simulation system.

- Programs based on XML are clear and understandable for the reader. Though the use of opening and closing tags in angle brackets can increase the length of a program, the diverse software tools allow the user to view and explore each component of the program separately as well as its general structure.
- XML does not require any particular operating system or software.
- XML syntax is familiar to the majority of computer programmers, and therefore it will be easier to understand and use the C-based action language for the new users.
- The existence of XML parsers simplifies the development of action languages based not only on C, but on almost any other popular language in the future.
- New features can be added to the language while preserving the backward compatibility by introducing new tags.
- Existing XML editors allow the user to check documents for correct syntax. Moreover, grammar of the action language can be easily expressed via XML Schema. The existence of such a schema will allow action programs to be validated for correct structure and semantics as well. Therefore, creating and debugging an action program becomes a simplified and automated process.

3.3.2 Proposed structure and syntax of the action language based on XML and C

The overall structure and grammar of the proposed action language are described through an XML Schema document [18]. This description is not a part of the preprocessor. However, it may be used to validate programs for correct syntax and structure. The schema document can also be beneficial for modifying and extending the language. The contents of the schema have been visualized with the help of XMLPad editor (see Section 4.1 for details).

Simple types

As the proposed action language is based on XML and C languages, the content of the program elements is generally fragments of C code to be included into C file without further translation. These fragments of code are described in the schema merely as XML strings. If code fragments contain characters that must be escaped, such as ampersand character (&) or the left angle bracket (<), they may be encapsulated into CDATA sections. A CDATA section begins with the delimiter <![CDATA[and ends with]]> delimiter. Characters between these delimiters, including markup characters, are interpreted as characters only. The existence of CDATA markup will not affect the way program is processed. Other types of content utilized in the schema include integer and decimal

values defined in [18], as well as the types called *nametype* and *liststype*. These types can be defined as follows:

```
<xs:simpleType name="nametype">
    <xs:restriction base="xs:string">
        <xs:pattern value="[A-Za-z_][A-Za-z_0-9]*"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="listtype">
    <xs:restriction base="xs:string">
        <xs:pattern value="[a-zA-Z0-9_, ]*"/>
</xs:restriction>
```

Each element belonging to *nametype* is intended to store single identifiers such as the name of an action or a class. Each element of type *listtype* contains a list of identifiers delimited by commas or whitespace characters. It can represent a list of parameters or participants of an action. Each case is illustrated further in this section.

Overall structure of a program

An XML-based action language program is a hierarchical tree of elements. The root element is marked by <*program*> tag. The root element is a complex element, the structure of which is described by Figure 9. The root element *program* contains a set of complex elements described below and two simple elements: *threads* and *types*. These elements may appear in any order, but the suggested order is illustrated by Figure 9. Each of these elements must appear at most once; *threads* and *types* elements may be omitted.

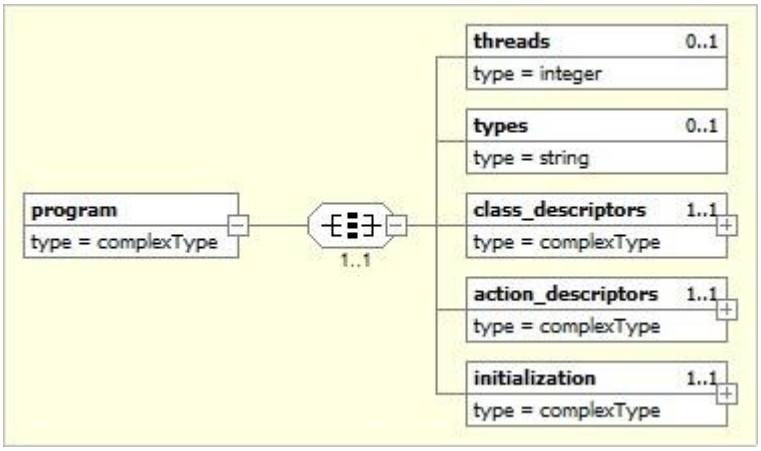

Figure 9: *Structure of the root element*

Element *threads* contains a positive integer value representing a number of executing threads. Current maximum of the simulation environment is 10.

Element *types* contains a fragment of C code describing constant values, type definitions and other information required for correct compilation of the C code fragments in the program.

All the class descriptors defined in the action program are accumulated into element *class_descriptors*, which is depicted on Figure 10. This element contains a list of *class_descriptor* elements. Each class descriptor must contain an element representing the name of the class, belonging to *nametype*. Another compulsory element *classtype* contains a name of a type, which is either of a basic C type or is defined in the *types* section. Optional elements of a class descriptor contain the information needed to produce functions to copy, destroy and initialize an object.

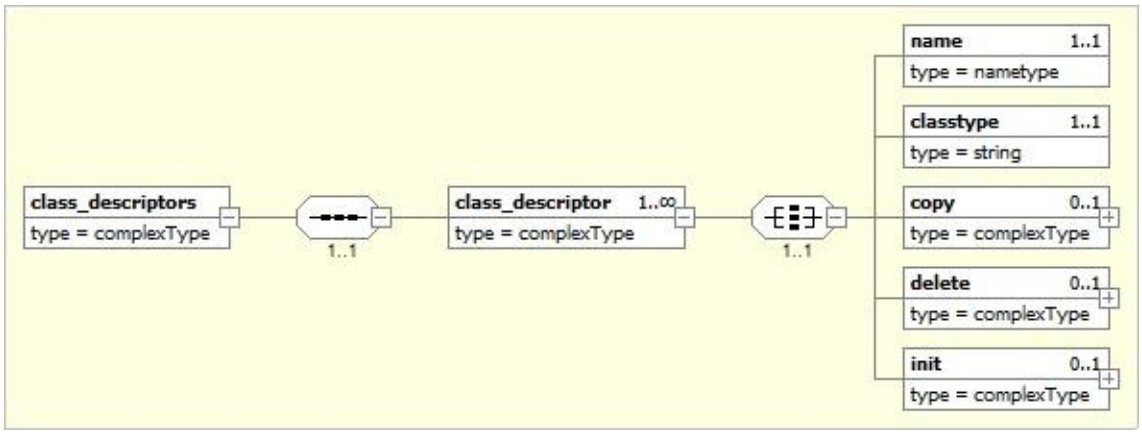

Figure 10: *Structure of the class_descriptors element*

The optional elements of a class descriptor are depicted on Figure 11. Element *copy* includes a fragment of code instructing how to make a copy of an object as well as identifiers used in this code to refer to the old and new objects. These identifiers are assumed to belong to the *classtype* type and are accessed as such. For example, if the *classtype* is a structure, it is accessed by the dot notation in the code. Element *delete* contains a fragment of code to be executed when an object of this class is deleted. Element *init* determines how the object is initialized after being created. It can either contain the value of the whole object or a set of sub-elements providing the individual values for distinct fields of a structure. Another way of initializing objects is specifying singular values for each object in the *initialization* element of the program.

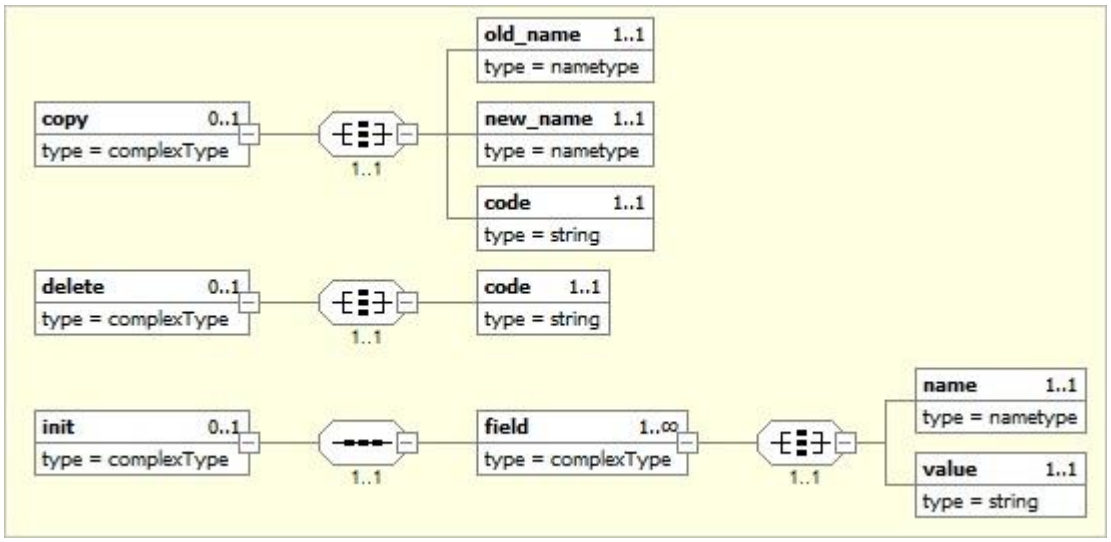

Figure 11: *Optional elements of a class descriptor*

The *action_descriptors* element presented in Figure 12 accommodates a list of action descriptors. Each *action_descriptor* contains a number of sub-elements. The *name* element defines a unique name for the descriptor. Another mandatory element for an action descriptor is *body* proving a fragment of code to be executed if the action is called. Optional elements include a list of parameters, a common guard, a relative deadline and a rate.

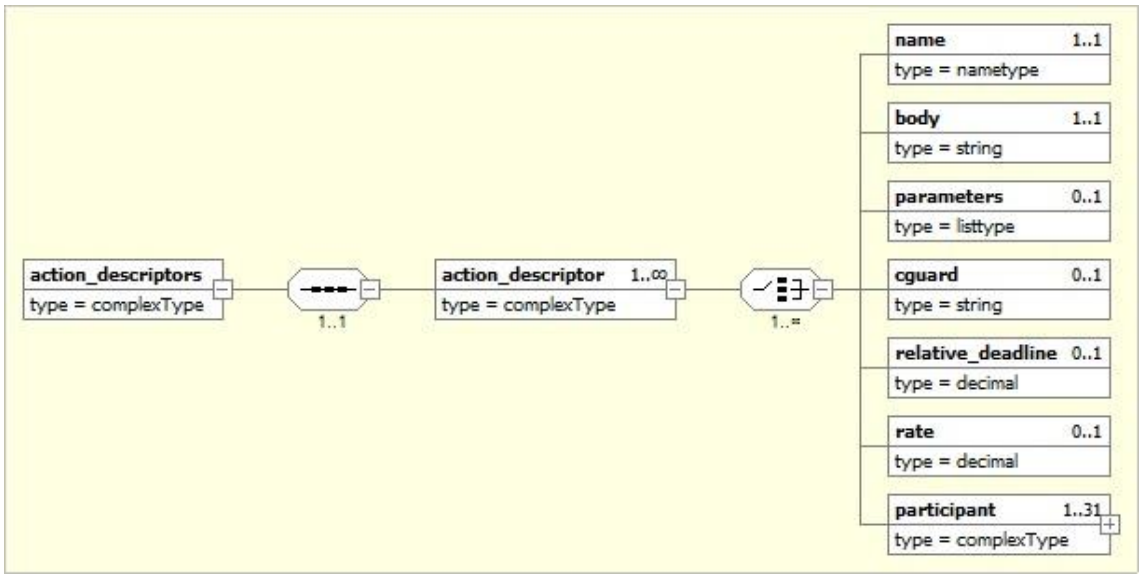

Figure 12: *The structure of the action_descriptors element*

The *parameters* element inside an action descriptor defines a list of formal names for parameters of the action descriptor, by which they are referred in guards and the body of the descriptor. Element *cguard* contains the Boolean expression corresponding to the common guard of the action. If the guard is omitted, it is assumed to be true. Elements *rate* and *relative_deadline* are included in the descriptor only if the common guard is

true or omitted. This is verified by the pre-processor, and not by the schema. These elements correspond to the timing settings, explained in Subsection 3.1.1.

Each action descriptor includes at least one *participant* element. The structure of such an element is illustrated by Figure 13. Two sub-elements are mandatory for each participant, defining its name and class. Element *name* specifies the formal name of the participant to be used in the local and common guards and the body. This name must be unique within a descriptor. Element *class* contains the name of the class the participant belongs to. There must exist a class descriptor with the same name within the *class_descriptors* element. Optional element *lguard* correspond to the local guard for this participant in a form of Boolean expression. If this element is omitted, it is assumed to be true. Elements *local_timeout*, *local_deadline* and *one_shot* may specify the local timing settings for this participant. These properties were discussed in Subsection 3.1.1.

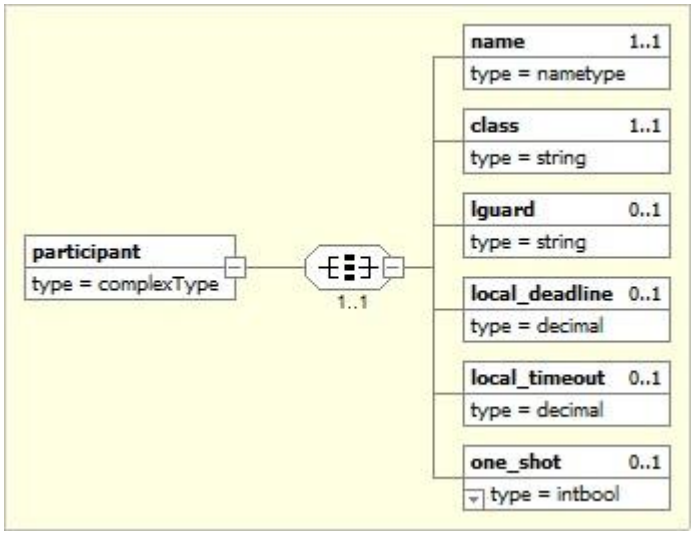

Figure 13: *The structure of a participant*

The structure of the *initialization* element is displayed in Figure 14. It consists of two mandatory elements - objects and actions. Element *objects* accommodates a list of objects that exist in the system. Element *actions* consists of a number of action elements, each of them describing an instance of a particular action descriptor.

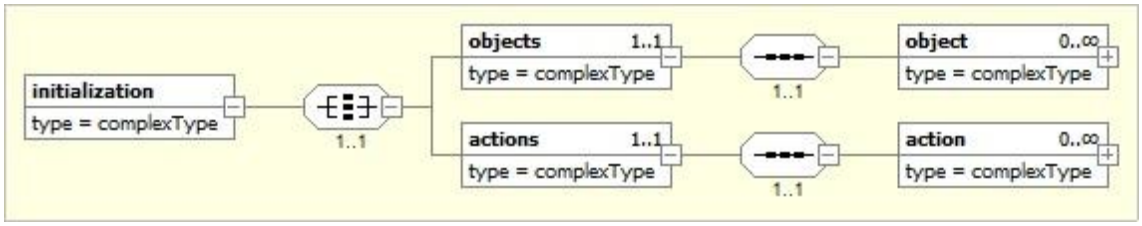

Figure 14: *Initialization element*

The structure of an object element is illustrated by Figure 15. Each object element represents the creation of not a single object, but a number of objects that belong to the

same class. Element *class* contains a single class name defined in the *class_descriptors* elements. Element *names* belonging to *listtype* must provide one or more names of the objects to be created. Each object must have a unique name to be referred within the *initialization* element. Optional element *initvalues* may determine a set of values separated by white spaces and optional commas to be assigned to the created objects. The number and the order of the individual values should be the same as in the corresponding names element. There should not be spaces inside each individual object value. If the class description includes the initialization instructions, they are overwritten by the *initvalues* contents inside the *initialization* element.

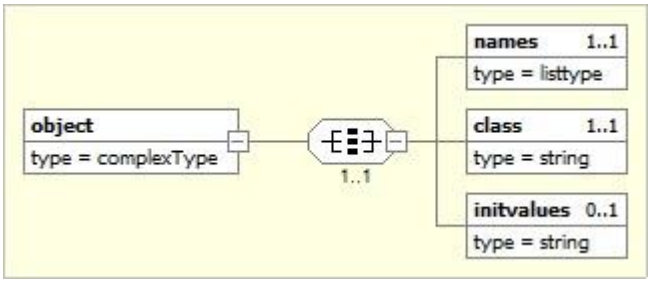

Figure 15: *The structure of an object element*

The structure of an element representing a single action is depicted in Figure 16. An action element must include a name of the descriptor defined among action descriptors as well as the list of participating objects. Element *with* of *listtype* must define such a list, containing object names separated by comma. These names must be defined within the *objects* element. The participants are assigned to the action in the same order as in the corresponding *action_descriptor* element. The optional element *parameters* of *listtype* contains the actual values of the action parameters. Their number and order is the same as in the *parameters* element of the corresponding action descriptor. The current version of the system considers parameters to be integer values.

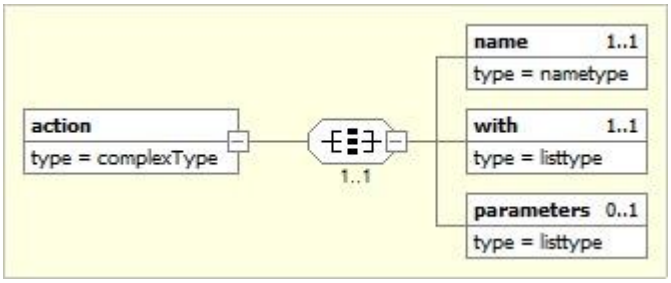

Figure 16: *The structure of an action element*

Besides the features of an action program described above, several specific function-like macros are introduced additionally. These macros provide the means of creating and deleting actions or objects from another action. They can be used inside the body of any action alongside the C code. Their signatures are described below:

```
CREATEACTION (action_descriptor_name,
```

```
    ...
    //uint parameters // This is repeated parameter_count times, 0 or more
    //object_indicator // this is repeated participant_count times, 1 or more
);
DELETEACTION();
DUPLICATE(Objectpointer new_objects[], uint count, ...);
```

The first actual parameter of the macro *CREATEACTION(...)* gives the name of the action descriptor, which instance is being created. Subsequent parameters should provide the values of action parameters determined by the corresponding action descriptor. The following one or more parameters specify the participating objects. If a participating object is one of the outer (creating) function participants, it is addressed by the name equal to the contents of the *name* element inside the corresponding *participant* element of the outer function. If a participating object has been created inside the same action body, the corresponding parameter of *CREATEACTION(...)* macro should be an identifier of type *Objectpointer*.

The macro *DELETEACTION()* provides the functionality for deleting an action from inside the body. In the current version of the proposed environment, only the executing action can be deleted, therefore, no parameters are needed. Empty parentheses are part of the macro for consistency purposes.

The macro *DUPLICATE(...)* allows to duplicate a set of objects. The *new_objects* array is created in the body before the use of the macro.

The examples of action programs using the macros defined above are provided in Section 4.3.

4. IMPLEMENTATION OF THE PRE-PROCESSOR

This chapter describes the implementation of the pre-processor that accepts programs written in the XML-based action language and produces C programs of the special structure understandable for the existing environment. The chapter is organized as follows. The software tools used for illustrative purposes in this thesis as well as for the implementation of the pre-processor are listed in Section 4.1. Section 4.2 describes the internal structure of the pre-processor and the meaning of the possible error messages. Several examples of the action systems designed by means of the proposed demonstration environment are presented in Section 4.3. Finally, possible ideas for the future work are proposed in Section 4.4.

4.1 Used software tools

The software solutions below have been used in this thesis for clarification and visualization purposes; they are not crucial for the utilization of the proposed system. Action program can be likewise developed in a simple text editor. Software tools used for illustration purposes include:

- XmlPad professional editor for XML (Apache License V2.0, Open Software License 3.0). Available: http://www.wmhelp.com/
- Notepad++ (GNU General Public License). Available: https://notepad-plus-plus.org/
- XML Notepad 2007 (Microsoft Public License). Available: http://www.microsoft.com/en-us/download/details.aspx?id=7973

As for the implementation of the pre-processor, RapidXml XML parser was used for parsing XML documents containing C-based action programs (available on http://rapidxml.sourceforge.net/).

RapidXml is a DOM XML parser written in C++. DOM (Document Object Model) is a platform- and language- independent specification which determines the interface for manipulating HTML and XML documents [2]. This specification presents a document as a as a set of objects organized into a hierarchical tree structure (DOM tree) and provides the mechanisms for accessing and updating them. Though RapidXml generally follows W3C DOM requirements, it contains several secondary incompatibilities. However, it succeeds in parsing all valid XML files in W3C conformance suite [10].

RapidXml was chosen for the implementation of the pre-processor for a number of reasons:

1. Parsing speed. RapidXml was created in an attempt to maximize the parsing speed. It demonstrates considerably faster performance compared to other popular parsers due to the use of multiple techniques such as storing pointers to the source text instead of copying strings [10].
2. Simplicity. The interface that the parser provides is clear and concise. It is determined through a header file, and the additional convenience and printing functions are described in two more header files. In order to use the parser in a C++ program, it is sufficient to download these header files and refer to them through *#include* directive.
3. Portability. RapidXml depends on a small subset of standard C++ library, which allows it to be compiled on any conventional compiler.
4. License terms. RapidXml is free software. The developer can choose between the MIT license and the Boost Software License in order to use the parser. These licenses allow to develop both free and proprietary software solutions based on RapidXml functionality as well as to modify the source code.

The RapidXml functions used in the implementation of the pre-processor include functions creating a hierarchical tree from the XML document and traversing this tree in the vertical and horizontal directions.

4.2 Implementation details

The pre-processor is implemented in C++. It has been compiled under Linux and Windows operating systems with GNU C++ Compiler (g++).

To compile the pre-processor code, the *makefile* provided with the environment can be invoked. Alternatively, the following line can be used:

```
g++ -Wall -o actionparser actionexception.cpp actionparser.cpp main.cpp
```

The pre-processor accepts two command line arguments. The first argument determines the name of the input file containing an action program, while the second one gives the name for the C code file to be created. After execution, the output file contains C code that can be compiled and executed using the simulation environment described in Section 3.1. The logical design of the pre-processor and the structure of the output file are explored below.

First, the necessary *#include* directives are included in the output program. Then, the auxiliary C code description is copied to output from the *types* element.

Next, the *class_descriptors* element is processed. The functions responsible for initializing, copying and deleting class instances are printed. The information about classes is saved into a map for later use.

Then, the details of action descriptors are saved into another map. For each action, local guards, the common guard and the body functions are printed. If any of the guards is omitted, the corresponding function is not created. The timings setting function is printed. If the rate or/and the relative deadline is specified for the action, the pre-processor checks that the common guard is omitted or true, then these settings are assigned in the timings setting function. The body code is scanned for macro functions calls (described in Subsection 3.3.2). Each macro call is expanded accordingly.

Furthermore, the application function is printed. The pre-processor iterates over the class map, generating a *create_class(...)* function call for each of the descriptors. Similarly, *create_action_descriptor(...)* function calls are printed for each of the action descriptors.

Next, the *initialization* element is parsed. The pre-processor examines each of the *object* elements to make sure that valid class names are given in the *names* sub-element of the object element. The *names* sub-element of each object element is parsed to retrieve individual object names and save their details into the object map.

Then, the action instances are processed. Each *action* element is checked for validity of the action name against the action descriptors map. The pre-processor verifies that the number of objects in the *with* sub-element corresponds to the number of formal action participants. Next, the list of parameters is checked to contain the names of existing objects only. The information about the action instances and their actual participants is saved into a vector.

Then, the functions creating objects and actions are generated. Each action including parameters is checked to have the number of actual parameters equal to the number of formal parameters in the corresponding action descriptor.

Finally, the main function of the program is generated. If the number of executing threads has not been specified in the *threads* element, the main function includes the request to read this number from stdin input.

As the action program in the proposed simulation environment included unprocessed fragments of C code, its correct execution cannot be guaranteed by the pre-processor. Moreover, to minimize the possibility of errors, each action program is advised to be validated against the schema for the action language described in Subsection 3.3.2. The list of possible errors detected by the pre-processor itself is provided below. The messages produced by the pre-processor are highlighted with italic. For each message to be printed if an error is found in the action program, the source of this error is explained:

1. *“Action descriptor should have at least one participant: _name”*. The *action_descriptor* element of name *_name* has no *participant* sub-elements.
2. *“Either rate and relative deadline or common guard should be provided: _name”*. If the rate or/and the relative deadline is defined for an action descriptor named *_name*, the common guard should be omitted or equal to true.
3. *“Initial values should be provided for particular fields: _name”*. If the *init* sub-element of a *_name* class descriptor provides initial values for a structure, each *field* sub-element should specify a *name* and a *value* sub-elements assigning the values to each field.
4. *“Name and class should be provided for each object”*. A sub-element *object* of the *initialization* element does not include the *names* or the *class* sub-element.
5. *“Name and type should be provided for each class"*. One of the *class_descriptor* elements does not include the *name* or the *type* sub-elements.
6. *“No action with this name exists: _name”*. The program attempts to create an action from an unknown action descriptor. It can happen inside one of the *action* sub-elements of the *initialization* element, or inside a *CREATEACTION(…)* macro call.
7. *“No actual participants provided: _name”*. The list of participating objects for one of the instances of *_name* action descriptor has not been provided.
8. *“No class with this name exists: _name”*. An element refers to the unidentified class. This can happen either inside a *participant* sub-element of an action descriptor, or inside an *object* sub-element of the *initialization* element.
9. *“No object with this name exists: _name”*. The sub-element *with* of an action contains an identified object of name *_name*.
10. *“Number of actual parameters doesn't match definition: _name”*. The sub-element *parameters* of an action element contains a number of values that differs from the number of formal parameters of the corresponding descriptor.
11. *“Number of actual participant doesn't match definition: _name”*. The sub-element *with* of an *action* element contains a number of object names that differs from the number of formal participants in the corresponding descriptor.
12. *“Number of inner function participants and/or parameters doesn't match definition: _name”*. In one of the *CREATEFUNCTION(...)* macro calls, the number of parameters or participants for the inner action of *_name* action descriptor does not match the corresponding action descriptor.
13. *“Objects for inner duplication not specified in action descriptor: _name”*. In one of the *DUPLICATE(...)* macro calls inside the body of *_name* action descriptor the parameters are provided incorrectly.
14. *“One of the objects doesn't have initial value: _name”*. The *initvalues* sub-element of an *object* element contains less initial values than there are objects in the corresponding *names* sub-element.
15. *“Tag contents are empty”*. The tag contents of an element are empty.

4.3 Demonstration programs in action language

4.3.1 The crossing example

To illustrate the possible use case of the proposed simulation system, the problem described in Section 2.4 is resolved using the action language based on XML and C.

The structure of the solution corresponds to the rules described in Section 2.4. The whole program with class descriptors, action descriptors, objects and actions collapsed is illustrated by Figure 17. The number of threads is chosen to be 3 without loss of generality.

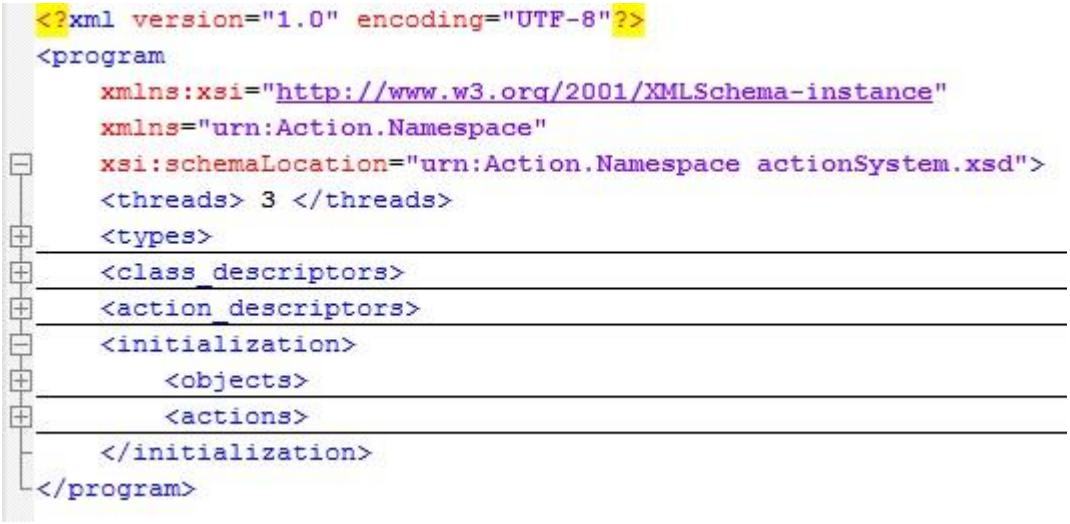

Figure 17: *The crossing program*

The set of class descriptors in the solution corresponds to the classes identified in Section 2.4 and is presented in Program 2. Objects of these classes are initialized in the *initialization* element, with the exception of class *Control*, which provides the initialization instructions within the class descriptor. The value of the *init* element of class *Control* does not have to be encapsulated into CDATA section since it does not contain any markup characters. However, it could contain markup characters as a C code fragment. The existence of CDATA markup does not affect the way the program is processed.

There is no need for copying or destroying instructions for any class.

```
<class_descriptors>
    <class_descriptor>
        <name>Lane</name><classtype>struct Lane</classtype>
        </class_descriptor>
    <class_descriptor>
        <name>Light</name><classtype>struct Light</classtype>
    </class_descriptor>
    <class_descriptor>
        <name>Control</name><classtype>struct Control</classtype>
        <init><field>
```

```
            <name>state</name>
            <value><![CDATA[NSW_SNE]]></value>
        </field></init>
    </class_descriptor>
</class_descriptors>
```

Program 2: *Element class_descriptors of the crossing program*

The list of action descriptors is illustrated by Figure 18. The meaning of these descriptors correspond to the pseudo-code solution described in Section 2.4.

	name	parameters	participant	cguard	body
1	set_green		participant (2)		body
2	set_yellow		participant		body
3	set_red		participant (2)		body
4	reserve	current_state	participant (5)	cont.state==current_	body

Figure 18: *List of action_descriptor elements defined in the system*

The code of the action descriptor *set_red* is presented in Program 3 to serve as an example of the textual representation of a single action. An important thing to be noticed in Program 3 is the fact that the participants are treated as variables of the corresponding types, and not as pointers, both in the body and in the guards. The textual values “true” and “false” can be used for Boolean variables, as the pseudo-Boolean type has been defined in the simulation environment.

```
<action_descriptor>
    <name>set_red</name>
    <participant>
        <class>Lane</class><name>lane</name>
        <lguard>lane.safe==true</lguard>
    </participant>
    <participant>
        <class>Light</class><name>light</name>
        <lguard>light.lamp==YELLOW</lguard>
    </participant>
    <body><![CDATA[
        light.lamp = RED;
        lane.safe = false;
        printf("Light %s set RED at %u\n", light.name, (uint)time(NULL));
    ]]></body>
</action_descriptor>
```

Program 3: *Action set_red*

In the *object* sub-element of *initialization* element a named object is created for each of the lanes and each of the traffic lights, as well as the single *control* object to switch between states. All the objects representing lanes and traffic lights are initialized on this phase. The element responsible for the creation of lanes is presented in Program 4.

```
<object>
    <class>Lane</class>
    <names>nsw, sne, ne, sw, wse, enw, es, wn</names>
```

```
    <initvalues><![CDATA[
        {"nsw",0}, {"sne",0}, {"ne",0}, {"sw",0}, {"wse",0} {"enw",0},
        {"es",0}, {"wn",0}
    ]]></initvalues>
</object>
```

Program 4: *Objects of class Lane*

The actions are created inside the *action* sub-element of the *initialization* element. A set of actions corresponding to a single lane *nsw* is shown on Program 5.

```
<action><name>set_red</name><with>nsw, tl_nsw</with></action>
<action><name>set_yellow</name><with>tl_nsw</with></action>
<action><name>set_green</name><with>nsw, tl_nsw</with></action>
```

Program 5: *Actions corresponding to nsw lane*

After the action program is processed with the pre-processor and the generated C-code is compiled with the simulation environment, it can be executed. The traffic system is not intended to terminate.

4.3.2 The ping-pong example

The simple action program presented in this subsection is intended to demonstrate the use of macro calls. It implements the ping-pong game between two players. The only logical entity recognized in this action system is a ball, which has a color and a state indicating at what side of the table the ball is now. For illustration purposes, it is assumed that the system is stopped after several rounds of the game.

The *types* and *class_descriptors* elements of the program are illustrated by Program 6. The auxiliary class *Counter* is intended to control the termination of the system.

```
<types><![CDATA[
    struct Ball {
        char color[20];
        enum {PING, PONG, FINISHED} state;
    };
]]></types>
<class_descriptors>
    <class_descriptor>
        <name>Ball</name><classtype>struct Ball</classtype>
    </class_descriptor>
    <class_descriptor>
        <name>Counter</name><classtype>int</classtype>
        <init>0</init>
    </class_descriptor>
</class_descriptors>
```

Program 6: *Class_descriptors and types elements of the ping-pong program*

There are three action descriptors to be defined in the system. First two action descriptors, *Ping* and *Pong*, are responsible for the actions of two players. The state of the

ball is alternated with the help of these actions, representing the bouncing of the ball. The third action descriptor, *Referee*, implements managing of the game. This action has two participants, first being the ball, and the second being the counter object. The action is executed only if the counter is equal to 0 or if it is greater than some specified number of rounds given as the action parameter. Initially, the counter is equal to 0, meaning that no rounds have been played yet. In this case, the game is started by creating actions for bouncing the ball. If the action is executed when the counter is greater than the value of the parameter, the executing action is deleted. Therefore, the system terminates. The action descriptors are demonstrated in Program 7.

```
<action_descriptors>
    <action_descriptor>
        <name>Ping</name>
        <participant><class>Ball</class><name>ball</name>
            <lguard>ball.state == PING</lguard></participant>
        <participant><class>Counter</class><name>counter</name></participant>
        <body><![CDATA[
            counter++;
            printf("%s ball: ping! round %i\n", ball.color, counter);
            ball.state = PONG;
        ]]></body>
    </action_descriptor>
    <action_descriptor>
        <name>Pong</name>
        <participant><class>Ball</class><name>ball</name>
            <lguard>ball.state == PONG</lguard></participant>
        <body><![CDATA[
            printf("%s ball: pong!\n", ball.color);
            ball.state = PING;
        ]]></body>
    </action_descriptor>
    <action_descriptor>
        <name>Referee</name>
        <parameters>rounds_number</parameters>
        <participant><class>Ball</class><name>ball</name></participant>
        <participant><class>Counter</class><name>counter</name></participant>
        <cguard>counter == 0 || counter >= rounds_number</cguard>
        <body><![CDATA[
            if (counter == 0) {
                printf("%s ball game started\n", ball.color);
                CREATEACTION(Ping, ball, counter);
                CREATEACTION(Pong, ball);
            } else {
                ball.state = FINISHED;
                printf("%s ball game finished\n", ball.color);
                DELETEACTION();
            }
        ]]></body>
    </action_descriptor>
</action_descriptors>
```

Program 7: *Action descriptors of the ping-pong program*

In the initialization section of the program, the objects representing a ball and a counter are created, as well as the single action of *Referee* descriptor, which will create and manage the game. The initialization element if illustrated by Program 8.

```
<initialization>
    <objects>
        <object>
            <class>Ball</class><names>red_ball</names>
            <initvalues>{"Red",PING}</initvalues>
        </object>
        <object><class>Counter</class><names>counter</names></object>
    </objects>
    <actions>
        <action>
            <name>Referee</name><parameters>5</parameters>
            <with>red_ball, counter</with></action>
    </actions>
</initialization>
```

Program 8: *Initialization element of the ping-pong program*

Although the action of *Referee* descriptor is created with parameter *rounds_number=5*, the number of rounds in the game can be different depending on the execution flow. When the compiled program is executed several times in a row, the number of rounds can differ from one execution to another, though always being greater than 5. The reason for this is the non-deterministic choice of the next action to be executed. When the counter is set to 5 by the *Ping* action, the *Referee* action is not executed immediately. The possible output of this action system is demonstrated on Program 9. If the system required the exact number of rounds to be played, it could be achieved by changing the design or adding timing settings to the action descriptors.

```
Red ball game started
Red ball: ping! round 1
Red ball: pong!
Red ball: ping! round 2
Red ball: pong!
Red ball: ping! round 3
Red ball: pong!
Red ball: ping! round 4
Red ball: pong!
Red ball: ping! round 5
Red ball: pong!
Red ball: ping! round 6
Red ball: pong!
Red ball: ping! round 7
Red ball game finished
Creation part 2, action found 18 action null 7474 queue actions 18
Action count 2, create_count 0 mutex called 18
Executor 2: 6 rounds (5 guards, 1 bodies, 0 deletes)
Executor 0: 6 rounds (2 guards, 4 bodies, 0 deletes)
Executor 1: 6 rounds (4 guards, 2 bodies, 0 deletes)
Common guard evaluated 5 times, local guard 32 times
```

Program 9: *Ping-pong program output*

4.4 Future ideas

The demonstration environment presented in this thesis is implemented on top of a conventional operating system. Therefore, the ensuing course of actions includes the hardware implementation of action computing and the development of the special operating system that includes a scheduler to organize the flow of actions. This will allow to take advantage of the built-in management of parallel computing provided by the action approach. However, the details of such implementation are beyond the scope of this thesis.

The immediate future work concerning this simulation environment can include implementing the remaining timing settings of an action. Additionally, more demonstration programs can be designed to illustrate the application of action approach for the implementation of well-known programming problems.

The proposed demonstration environment is intended to facilitate the design of action systems. Compared to the simulation framework described in Section 3.1, it expresses the concepts of the action paradigm in a more comprehensible way. Nevertheless, the proposed environment lacks instruments for design, debugging and visualization of action-based solutions. In response to this problem, an integrated development environment (IDE) can be implemented based on the proposed demonstration system.

The code editor can be implemented as a part of the IDE for action systems. It can hide the XML tags from the user and present an action program in a clear and shortened way. Moreover, the suggested IDE can include a browser of class descriptors, a browser of action descriptors, an action browser and an object browser. A diagram illustrating dependencies between actions and objects can also be available in the IDE. As humans tend to think sequentially, understanding action paradigm can be a challenging task. The suggested visual tools can be beneficial for analyzing the structure of an action program.

Figure 19 portrays a possible diagram of dependencies existing in the action system that corresponds to the radio example described in Section 2.3. The system includes a pair of radios capable of transmitting messages to each other through a channel. This diagram illustrates the relationships between the elements of the system. It can be helpful for an inexperienced user as well for a well-versed programmer in case of larger systems.

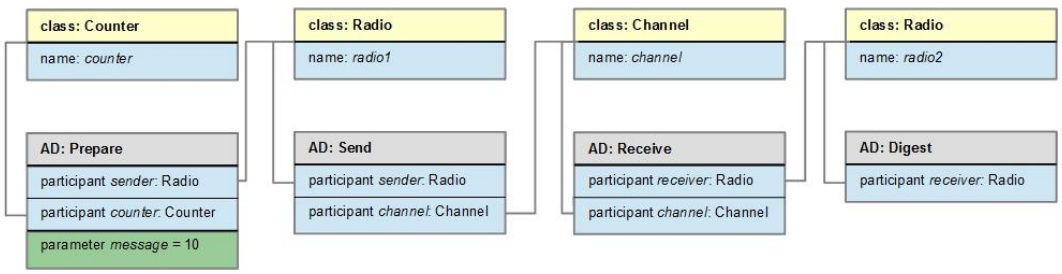

Figure 19: *Dependencies Diagram for the Radio example*

The existence of debugging tools is necessary for the user-friendly design of action programs. A programmer would be able to observe a list of objects and actions existing in the system at the given moment, as well as the contents of the action queue. Since each action is an atomic unit of execution, debugging could include action by action stepping, but an action cannot be interrupted.

The flow of execution can be illustrated by the diagram containing the list of processing units and the list of actions executed by each of them. An example of such a diagram is demonstrated on Figure 20. This example is based on the problem presented in Section 2.4. This diagram is intended to show the user how the scheduler manages the execution of actions and distributes them between processing units. The user can also view a list of participating objects by maximizing action rectangles. The purpose of the diagram on Figure 20 is to illustrate the concept; the actual flow of execution depends on the implementation of the scheduler.

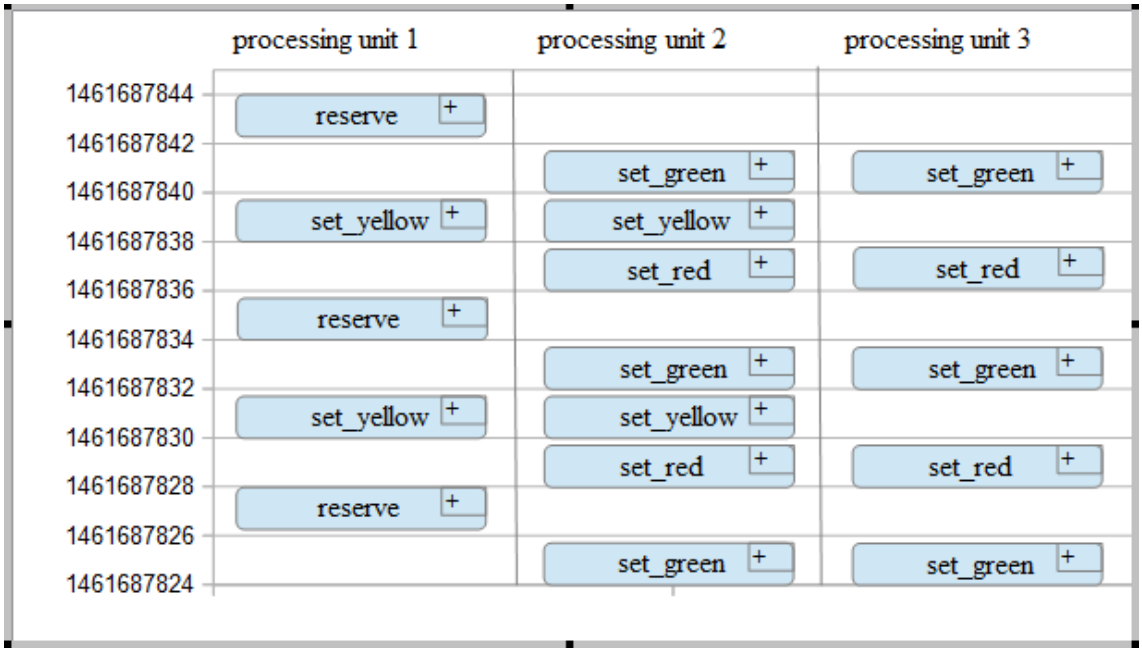

Figure 20: *Diagram representing flow of execution*

5. CONCLUSIONS

In this work, a simulation environment for demonstrating and exploring action paradigm has been presented. This environment is based on the existing framework simulating action execution. Action systems have to be represented by C programs of a particular structure to be used in this simulator. The pre-processor proposed in this thesis accepts a program written in an action language and produces a C program to be executed in the existing framework. Moreover, the action language to be used in this environment has been designed. Since the existing framework requires the use of C programs, the functional parts of the proposed language are fragments of C code. The hierarchical structure of an action program is represented through adapting XML syntax for the language. The pre-processor does not perform the full translation of a program. It restructures the action program and generates C code understandable for the existing framework.

The features and regulations of a tentative action language are explored in this thesis work. The interface of the existing environment is described in order to establish the necessary components of the language to be designed. The benefits of the XML syntax for this language are investigated, and this syntax becomes a base for the action language. The description of the designed action language is presented next. As this language is based on XML syntax, it is illustrated by a corresponding XML Schema.

The pre-processor is implemented in C++ using the existing XML parser for initial processing of the action program. The internal logic of the pre-processor and the meaning of the error messages a programmer may encounter are explained. Detailed examples are used to illustrate the implementation of action systems through the proposed environment.

The proposed system allows to examine and analyze action approach by designing and testing programs in the action language. The environment can be helpful for programmers who want to familiarize themselves with action paradigm. The system is beneficial for these purposes since it does not require learning an exclusively unfamiliar language. As C and XML are well-known among computer experts, it is relatively easy to adapt to the action language suggested in this thesis.

However, this environment cannot utilize many benefits of the action paradigm as it is based on threads. The existing operating system does not make is possible to make sensible measurements on efficiency of the system. The possible future work includes de-

sign of the special operating system featuring a scheduler to manage the execution of actions.

The possible future work concerning the demonstration environment can include the implementation of the specialized action-oriented integrated development environment that will allow to visualize, analyze and debug action programs. The existence of these tools could widen the community of potential action programmers and facilitate the design of large-scale action systems.

REFERENCES

[1] K. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1988.

[2] Document Object Model (DOM) Requirements, W3C Working Group Note, 2004. Available: http://www.w3.org/TR/2004/NOTE-DOM-Requirements-20040226/

[3] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, 2008. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[4] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers, 2008.

[5] H.-M. Järvinen, Actions, Objects, and Subjects, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2013, pp. 285-291.

[6] H.-M. Järvinen, Experimental action environment, 2015.

[7] H.-M. Järvinen, R. Kurki-Suonio, The DisCo Language, Report 8, Tampere University of Technology, 1990.

[8] H.-M. Järvinen, R. Kurki-Suonio, The DisCo Language and Temporal Logic of Actions, Report 11, Tampere University of Technology, 1990.

[9] H.-M. Järvinen, The Design of a Specification Language for Reactive Systems. Doctoral thesis, Tampere University of Technology, 1992.

[10] M. Kalicinski, RAPIDXML Manual, 2009. Available: http://rapidxml.sourceforge.net/manual.html

[11] R. Kurki-Suonio, A Practical Theory of Reactive Systems, Springer, 2005.

[12] R. Kurki-Suonio, H.-M. Järvinen, Action System Approach to the Specification and Design of Distributed Systems, Report 5, Tampere University of Technology, 1989.

[13] R. Kurki-Suonio, T. Kankaanpää, On the Design of Reactive Systems, Report 1, Tampere University of Technology, 1988.

[14] G. S. Palani, Investigate current XML tools, IBM, 2011. Available: http://www.ibm.com/developerworks/library/x-xmltools/

[15] A. Silberschatz, P. Galvin, G. Gagne, Operating Systems Concepts, Wiley, 2005, pp. 96-102.

[16] A. Tanenbaum, Modern Operating Systems, Pearson, 2009, pp. 115-142.

[17] N. Walsh, A Technical Introduction to XML, ArborText Inc., 1988. Available: http://nwalsh.com/docs/articles/xml/

[18] XML Schema Definition Language (XSD), W3C Recommendation, 2012. Available: http://www.w3.org/TR/xmlschema11-1/. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/